Lecture 7: Template Functions

CS 106L, Fall ‘20

Today’s Agenda

- Recap: Iterators
- Template functions
- Announcements
- (supplemental materials) Variadic Templates
- Concept Lifting

Recap: Iterators

Iterators allow iteration over any container

whether ordered or unordered

STL Iterators

Generally, STL iterators support the following operations:

```
std::set<T> s;
auto iter = s.begin();
iter++;                          // increment; prefix operator is faster (why?)
*iter;                        // dereference iter to get curr value
(iter != s.end());                 // equality comparison

iter = another_iter;             // copy construction
```

STL collections have the following operations:

```
s.begin();                    // an iterator pointing to the first element
s.end();                         // one past the last element
```

Printing all elements in these collections

```
std::set<int> set {3, 1, 4, 1, 5, 9};
for (initialization; termination-condition; increment) {
  const auto& elem = retrieve-element;
  cout << elem << endl;
}

std::map<int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (initialization; termination-condition; increment) {
  const auto& [key, value] = retrieve-element;  // structured binding!
  cout << key << ":" << value << endl;
}
```

Printing all elements in these collections

```
std::set<int> set {3, 1, 4, 1, 5, 9};
for (auto iter = set.begin(); iter != set.end(); ++iter) {
  const auto& elem = *iter;
  cout << elem << endl;
}

std::map<int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (auto iter = map.begin(); iter != map.end(); ++iter) {
  const auto& [key, value] = *iter; // structured binding!
  cout << key << ":" << value << endl;
}
```

Another option: for-each loops!

For-each loops use iterators under the hood!

```
std::set<int> set {3, 1, 4, 1, 5, 9};
for (const auto& elem : set) {
  cout << elem << endl;
}

std::map<int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (const auto& [key, value] : map) {
  cout << key << ":" << value << endl;
}
```

🤔 Questions? 🤔

Template Functions

Sidenote: Ternary Operator

```
int my_min(int a, int b) {
    return a < b ? a : b;
}
```

if condition

return if true

return if false

```
// equivalently

int my_min(int a, int b) {

    if (a < b) return a;

    else return b;

}
```

Can we handle different types?

```
int main() {
    auto min_int = my_min(1, 2);
    auto min_name = my_min("Nikhil", "Ethan");
}
```

One way: overloaded functions

```
int my_min(int a, int b) {
    return a < b ? a : b;
}

std::string my_min(std::string a, std::string b) {
    return a < b ? a : b;
}
```

One way: overloaded functions

```
int my_min(int a, int b) {
    return a < b ? a : b;
}

std::string my_min(std::string a, std::string b) {
    return a < b ? a : b;
}
```

Bigger problem: how do we handle user-defined types?

We now have a generic function!

```
template <typename T>
T my_min(T a, T b) {
  return a < b ? a : b;
}
```

Template function syntax analysis

Declares the next declaration is a template

Specifies T is some arbitrary type

List of template arguments

```
template <typename T>
T my_min(T a, T b) {
  return a < b ? a : b;
}
```

Note: Scope of template argument T is limited to this one function!

Just in case we don't want to copy T

```
template <typename T>
T my_min(T a, T b) {
  return a < b ? a : b;
}
```

```
template <typename T>
T my_min(const T& a, const T& b) {
  return a < b ? a : b;
}
```

Live Code Demo:

Templates: syntax and initialization

There are two ways to call template functions!

```
template <typename T>
T my_min(const T& a, const T& b) {
  return a < b ? a : b;
}
```

Way 1: Explicit instantiation of templates

Compiler replaces every T with `string`

```
template <typename T>
T my_min(const T& a, const T& b) {
  return a < b ? a : b;
}

my_min<string>("Nikhil", "Ethan");
```

Explicitly states `T = string`

Way 2: Implicit instantiation of templates

Compiler replaces every T with int

```
template <typename T>
T my_min(const T& a, const T& b) {
  return a < b ? a : b;
}

my_min(3, 4);
```

Compiler deduces T = int

Be careful: type deduction can't read your mind!

Compiler replaces every T with char*

```
template <typename T>
T my_min(const T& a, const T& b) {
  return a < b ? a : b;
}

my_min(“Nikhil”, “Ethan”);
```

Comparing pointers -- not what you want!

Compiler **deduces** T = `char*` (C-string)

Our function isn't technically correct

```
template <typename T>
T my_min(const T& a, const T& b) {
  return a < b ? a : b;
}

my_min(4, 3.2);
// this returns 3
```

Compiler **deduces** T = `int`

Be careful: type deduction can't read your mind!

```
template <typename T, typename U>
auto my_min(const T& a, const U& b) {
  return a < b ? a : b;
}

my_min(4, 3.2);
// this returns 3.2
```

The return type is kind of complicated, so let the compiler figure it out

Accounting for the fact that the types could be different

You can overload non-template special cases

```
char* my_min(const char*& a, const char*& b) {
  return std::string(a) < std::string(b) ? a : b;
}

template <typename T, typename U>
auto my_min(const T& a, const U& b) {
  return a < b ? a : b;
}
```

If we get C-strings, run this special case

Otherwise, create a template

Template Instantiation: creating an “instance” of your template

When you call a template function, either:

- for explicit instantiation, compiler finds the relevant template and creates that function in the executable.
- for implicit instantiation, compiler looks at all possible overloads (template and non-template), picks the best one, deduces the template parameters, and **creates that function in the executable.**
- **After instantiation, the compiled code looks <u>as if</u> you had written the instantiated version of the function yourself.**

Template functions are not functions

They're a recipe for generating functions via instantiation.

~~🤔~~ Questions? ~~🤔~~

Announcements

Assignment 1 Will Be Released Tomorrow!

- Due Sunday, October 25 on Paperless
- There will be a very small warm-up due next week
- We’ll send out an announcement with all logistical details

Supplemental Material: Variadic Templates *

* not essential information to understand to make it through the rest of this class -- we're including this topic just because it's cool :)

How can we make this function even more generic?

```
int main() {
  auto min1 = my_min(4.2, -7.9);



}
```

Say, an arbitrary number of parameters?

```
int main() {
  auto min1 = my_min(4.2, -7.9);
  auto min2 = my_min(4.2, -7.9, 8.223);
  auto min3 = my_min(4.2, -7.9, 8.223, 0.0);
  auto min4 = my_min(4.2, -7.9, 8.223, 0.0, 1.753);
}
```

Take a moment to think about this

How would you write a recursive version of `my_min` that accepts a `vector`?

```
// assume nums is non-empty and T is comparable
template <typename T>
T my_min(vector<T>& nums) {






}
```

Take a moment to think about this

How would you write a recursive version of `my_min` that accepts a `vector`?

```
// assume nums is non-empty and T is comparable
template <typename T>
T my_min(vector<T>& nums) {
  T elem = nums[0];
  if (nums.size() == 1) return elem;
  auto min = my_min(nums.subList(1));
  if (elem < min) min = elem;
  return min;
}
```

Let’s translate this into a variadic template!

Variadic templates can use compile-time recursion

```
template <typename T, typename ...Ts>
auto my_min(T num, Ts... args) {
  auto min = my_min(args...);
  if (num < min) min = num;
  return min;
}
```

Let's translate this into a variadic template!

Variadic templates can use compile-time recursion

```
template <typename T, typename ...Ts>
auto my_min(T num, Ts... args) {
  auto min = my_min(args...);
  if (num < min) min = num;
  return min;
}
```

Parameter pack: 0 or more types

Pack expansion: comma-separated patterns

Parameter Pack expansion examples

```
// expands to f(&E1, &E2, &E3)
f(&args...);

// expands to f(n, ++E1, ++E2, ++E3);
f(n, ++args...);

// expands to f(++E1, ++E2, ++E3, n);
f(++args..., n);

// f(const_cast<const E1*>(&X1), const_cast<const E2*>(&X2), ...)
f(const_cast<const Args*>(&args)...);
```

What does this expand to?

```
f(h(args...) + args...);

// f(h(E1,E2,E3) + E1, h(E1,E2,E3) + E2, h(E1,E2,E3) + E3)
```

Don't forget a base case!

```
template <typename T, typename ...Ts>
auto my_min(const T& num, Ts... args) {
  auto min = my_min(args...);
  if (num < min) min = num;
  return min;
}

template <typename T>
auto my_min(const T& num) {
  return num;
}
```

Calls to the function are pattern-matched to these templates

This pattern-matching happens at **compile time**, not runtime

```
template <typename T, typename ...Ts>
pair<T, T> my_min(T num, Ts... args) {
  auto [min, max] = my_min(args...);
  if (num < min) min = num;
  if (num > max) max = num;
  return {min, max};
}
```

Calls to the function are pattern-matched to these templates

This pattern-matching happens at **compile time**, not runtime

```
template <typename T, typename ...Ts>
pair<T, T> my_min(T num, Ts... args) {
}
```

```
auto [min3, max3] = my_min(4.2, -7.9, 8.223, 0.0);

// T = int
// Ts = int, int, int

// num = 4.2
// args = -7.9, 8.223, 0.0
```

The parameter pack is expanded into a comma-separated list

This pattern-matching happens at **compile time**, not runtime

```
template <typename T, typename ...Ts>
pair<T, T> my_min(T num, Ts... args) {
  auto [min, max] = my_min(args...);
```

```
Equivalent to:
auto [min3, max3] = my_min(-7.9, 8.223, 0.0);

Since:
// num = 4.2
// args = -7.9, 8.223, 0.0
```

```
}
```

The types do not have to be homogeneous

Example: A simplified version of C's `printf` function without format flags *

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  int pos = format.find('%');
  if (pos == string::npos) return;
  cout << format.substr(0, pos) << value;
  printf(format.substr(pos+1), args...);
}

void printf(string format) {
  cout << format;
}
```

* no worries if you're not familiar with the `printf` function. You'll learn about this if you take CS 107! :)

At compile time, the compiler instantiates the templates

First, using template deduction, we deduce T and Ts

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  …
  printf(format.substr(pos+1), args...);
}
```

```
printf("Lecture %: % (Week %)", 7, "Templates"s, 4);
```

At compile time, the compiler instantiates the templates

A new function is generated that matches our function call . . .

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  …
  printf(format.substr(pos+1), args...);
}
```

```
printf("Lecture %: % (Week %)", 7, "Templates"s, 4);

printf<int, string, int> // this function is generated
```

At compile time, the compiler instantiates the templates

…and everything is replaced with the instantiated types!

```
template <typename T, typename ...Ts>
void printf(string format, int value, string arg1, int arg2) {
  …
  printf(format.substr(pos+1), arg1, arg2);
}
```

```
printf("Lecture %: % (Week %)", 7, "Templates"s, 4);

printf<int, string, int> // this function is generated
```

At compile time, the compiler instantiates the templates

The recursive call tells us the next instantiation that we need

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  …
  printf(format.substr(pos+1), args...);
}
```

```
printf<int, string, int>

printf<string, int> // this function is generated next
```

At compile time, the compiler instantiates the templates

Instantiation again replaces the types

```
template <typename T, typename ...Ts>
void printf(string format, string value, int arg1) {
   …
   printf(format.substr(pos+1), arg1);
}
```

```
printf<int, string, int>

printf<string, int> // this function is generated next
```

At compile time, the compiler instantiates the templates

Again, the recursive call tells us the next instantiation that we need

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  …
  printf(format.substr(pos+1), args...);
}
```

```
printf<int, string, int>

printf<string, int>

printf<int> // this function is generated next
```

At compile time, the compiler instantiates the templates

Finally, the parameter pack is empty

```
template <typename T, typename ...Ts>
void printf(string format, string value) {
  …
  printf(format.substr(pos+1));
}
```

```
printf<int, string, int>

printf<string, int>

printf<int> // this function is generated next
```

At compile time, the compiler instantiates the templates

One more function is compiled: the non-template base function

```
void printf(string format) {
  cout << format;
}
```

```
printf<int, string, int>

printf<string, int>

printf<int>

printf // this function is generated next
```

At compile time, these functions are compiled

```
printf<int, string, int>

printf<string, int>

printf<int>

printf
```

Concept lifting

What assumptions are we making about the parameters?

Can we solve a more general problem by relaxing some of the constraints?

Why write generic functions?

Count the number of times **3** appears in a **std::list<int>**.
Count the number of times **“X”** appears in a **std::istream**.
Count the number of times **a vowel** appears in a **std::string**.
Count the number of times **a college student** appears in a **census**.

Remove as many assumptions as you can

How many times does an int appear in a vector of ints?

```
int count_occurrences(const vector<int>& vec, int val) {
  int count = 0;
  for (size_t i = 0; i < vec.size(); i++) {
    if (vec[i] == val) count++;
  }
  return count;
}

vector<int> v; count_occurrences(v, 5);
```

🤔 What is an assumption we're making here? (Type in the chat.)

How many times does an int appear in a vector of ints?

```
int count_occurrences(const vector<int>& vec, int val) {
  int count = 0;
  for (size_t i = 0; i < vec.size(); i++) {
    if (vec[i] == val) count++;
  }
  return count;
}

vector<int> v; count_occurrences(v, 5);
```

🤔 What if we want to generalize this beyond ints?

How many times does a <T> appear in a vector<T>?

```
template <typename DataType>
int count_occurrences(const vector<DataType>& vec, DataType val) {
  int count = 0;
  for (size_t i = 0; i < vec.size(); i++) {
    if (vec[i] == val) count++;
  }
  return count;
}

vector<string> v; count_occurrences(v, “test”);
```

🤔 Perfect! But what if we want to generalize this beyond a vector?

One possibility...

```
template <typename Collection, typename DataType>
int count_occurrences(const Collection& arr, DataType val) {
  int count = 0;
  for (size_t i = 0; i < arr.size(); i++) {
    if (arr[i] == val) count++;
  }
  return count;
}

vector<string> v; count_occurrences(v, “test”);
```

🤔 What is wrong with this? (Type in the chat.)

🚫 **The collection may not be indexable.** How can we solve this?

How many times does a <T> appear in an iterator<T>?

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
  int count = 0;
  for (initialization; end-condition; increment) {
    if (retrieval == val) count++;
  }
  return count;
}

vector<string> v; count_occurrences(arg1, arg2, “test”);
```

🤔 **Practice by filling in the blanks in the chat!**

How many times does a <T> appear in an iterator<T>?

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (*iter == val) count++;
  }
  return count;
}

vector<string> v; count_occurrences(v.begin(), v.end(), "test");
```

🤔 Great!

📝 We manually pass in **begin** and **end** so that we can customize our search bounds.

Live Code Demo:
Count Occurrences

We can now solve these questions...

Count the number of times **3** appears in a **list<int>**.
Count the number of times **'X'** appears in a **std::deque<char>**.
Count the number of times **'Y'** appears in a **string**.
Count the number of times **5** appears in the **second half of a vector<int>**.

But how about this?

Count the number of times **an odd number** appears in a **vector<int>**.
Count the number of times **a vowel** appears in a **string**.

🤔 Questions? 🤔

Recap

- **Template functions**
  - lets you declare functions that can accept different types as parameters!
- **(Supplemental) Variadic templates**
  - lets you declare functions that can accept an arbitrary number of parameters!
- **Concept lifting**
  - technique that we use to see how to generalize our code!

Next time:

lambda functions and algorithms